УДК 004.4'242

БУЗОВСКИЙ О.В.,
АЛЕЩЕНКО А.В.,
ПОДРУБАЙЛО А.А.

СИСТЕМА АВТОМАТИЧЕСКОЙ ГЕНЕРАЦИИ КОДОВ ПО ГРАФИЧЕСКИМ СХЕМАМ АЛГОРИТМОВ

В статье рассматриваются построение графических схем алгоритмов, процесс генерации кодов по графическим схемам алгоритмов. Как практическая сторона реализована система построения графических схем, предназначенных для дальнейшей генерации программных кодов. Программа позволяет создавать графические схемы, редактировать, сохранять и открывать их, а также генерировать соответствующие схемам программные коды и выдавать результаты их выполнения.

In article construction of algorithm graphical scheme, process of generation code of a program by graphical scheme is treated. As the practical side system of building graphical scheme for generation code of a program is realized. The software product makes it possible to create algorithm graphical schemes, edit, save and open they, to generate code of a program and to show results of their execution.

Целью работы является разработка приложения, реализующего построение графического представления программных функций [1] и последующую генерацию кода в заданном подмножестве языков программирования (в частности Java, Pascal, C). Основными функциями приложения являются следующие:

1. Построение изображения графических схем алгоритмов (ГСА).
2. Структурный, синтаксический и частично семантический контроль.
3. Генерация программного кода.

При проектировании редактора в качестве аналогов рассматривались следующие программные средства: графический редактор в Microsoft Office Word [2], Microsoft Office Visio, Kivio [3], Блоксхематор [4], Редактор блок-схем алгоритмов [5], Редактор блок-схем [6]. Основными недостатками данных программных средств являются:

1. Отсутствие возможности генерации кодов (кроме двух последних средств).
2. Отсутствие контроля ГСА.
3. Отсутствие специализации (Word, Visio, Kivio).

Перечисленные недостатки делают актуальным разработку собственного редактора.

При генерации кода основная сложность связана с проблемами типизации переменных. Данная проблема может рассматриваться в двух аспектах:

- отображение типа данных в окне редактора,
- автоматическое связывание [7].

В системе в данный момент реализовано, как отображение данных в окне редактора, так и частично, автоматическое связывание. Для выполнения функций автоматического связывания используется анализ кода (парсер), описанный ниже.

Внутреннее представление графической схемы алгоритма. Абстрактный компонент граф-схемы

Для унификации различных операций, проводимых над всеми компонентами граф-схем алгоритмов, как то прорисовка, сохранение, выделение компонент, была выделена абстракция, содержащая общие для различных типов элементов ГСА свойства и методы их обработки. В разработанной программной системе такому абстрактному компоненту соответствует абстрактный класс AbstractComponent (см. UML-диаграмму на рис. 1).

Свойство serialVersionUID присутствует для полного соответствия требованиям интерфейса Serializable, что в свою очередь не обходимо для корректного сохранения объектного представления ГСА в файл и дальнейшего его восстановления.

Поле blockProperties предназначено для хранения смысловых свойств компонент граф-схемы. Класс Properties, использованный в данном случае, позволяет хранить любые объекты, при этом для доступа используются объекты-ключи. В разработанной системе было решено использовать в качестве

ключей строки с названиями свойств. Строки определяются в качестве закрытых неизменяемых полей (в данном классе – это свойство SELECTED), а доступ к хранимым значениям осуществляется с помощью соответствующих методов чтения и записи (таких, как isSelected и setSelected). Таким образом значительно снижается вероятность выполнения каких-либо некорректных действий с данными, хранимыми внутри blockProperties.

Абстрактный метод draw добавлен для дальнейшей реализации прорисовки конкретного элемента в классах-наследниках.

Класс AbstractComponent используется в методах, ориентированных на обработку всех составляющих блок-схемы, как блоков, так и связей между ними.

Абстрактный блок

Данная абстракция выделена для унификации операций и свойств различных типов блоков граф-схемы алгоритма. Узлы граф-схемы различных типов имеют значительно больше общих между собой свойств, нежели те же узлы и соединения между ними. Соответственно класс AbstractBlock, реализующий в программном продукте абстракцию блока, имеет значительное число полей и методов доступа к ним.

Каждый блок, помимо свойства выделяемости, определенного в классе AbstractComponent, имеет абсциссу и ординату центра, длину, ширину, строку содержащегося в нем оператора и два соединения с другими блоками – входящее и исходящее. Также предусмотрено хранение структуры данных, образованной в результате лексического анализа содержащегося значения. Все вышеперечисленные свойства хранятся в закрытых полях, и доступ к ним возможен только с помощью специально определенных в этом классе методов по такому же механизму, как и тот, который был описан в классе AbstractComponent. Также в классе описаны методы, позволяющие, исходя из этих, непосредственно хранимых, свойств, вычислить координаты вершин прямоугольника, содержащего в себе данный блок (т.е. касающийся блока всеми четырьмя сторонами).

Абстрактный блок содержит два метода перемещения: moveTo, перемещающий центр узла в точку с абсолютно заданными координатами, и moveByOffSet, выполняющий перемещение блока в соответствии с заданным смещением.

Так же в этом классе реализованы специфицированные в абстракции компонента методы contains и isSelected. Первый проверяет, принадлежит ли точка с абсолютно заданными координатами данному элементу блок-схемы, второй определяет, попадает ли данный блок в прямоугольную область выделения, задаваемую в аргументах. Также реализован вариант метода contains, определяющий, принадлежит ли точка данной фигуре с некоторым допуском.

Абстрактным в данном классе остается лишь метод draw, призванный определять прорисовку конкретного компонента.

Диаграмма классов внутреннего представления ГСА приведена на рис. 2.

Модель алгоритма

Модель алгоритма – это внутреннее представление графической схемы алгоритма в целом. Класс AlgorithmModel, реализующий абстракцию модели ГСА, содержит ссылки на все экземпляры компонент, входящих в данную блок-схему. Класс соответствует требованиям интерфейса Serializable, что позволяет его запись в бинарные объектные потоки с последующим сохранением в файл.

Модель алгоритма обладает широкой функциональностью, позволяющей обеспечить все необходимые для работы с ГСА операции.

Диаграмма классов внутреннего представления ГСА приведена на рис. 1 и рис.2.

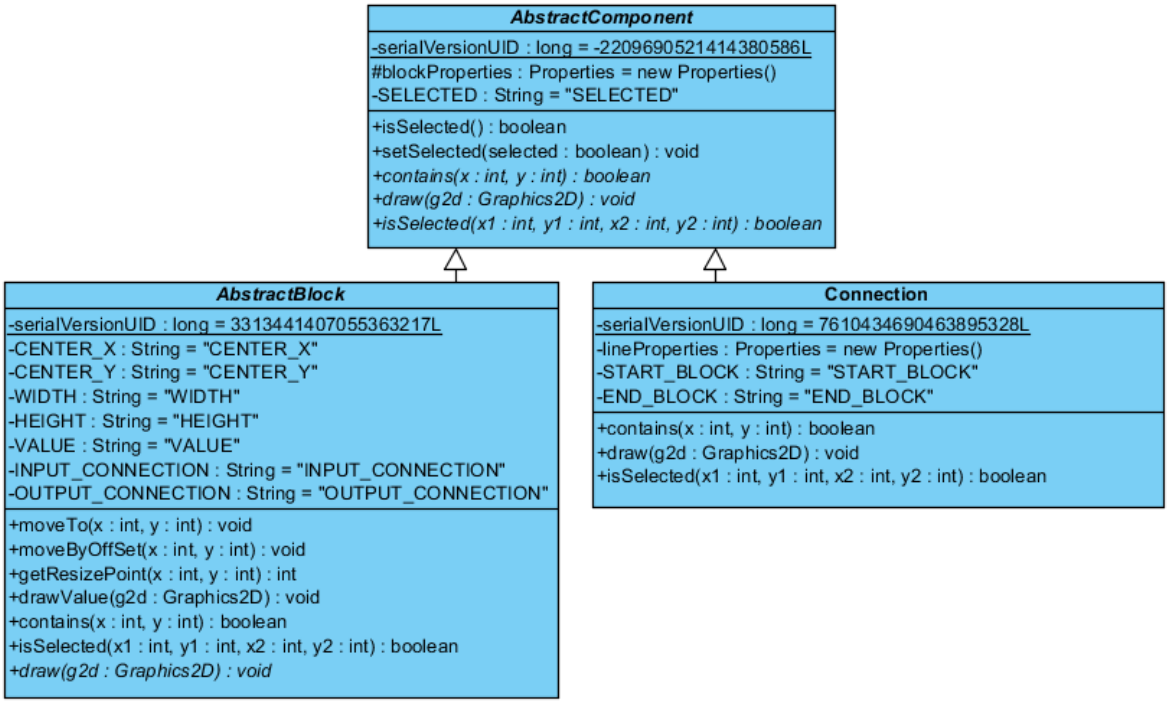

Рис. 1. Диаграмма классов-абстракций блоков и связей между ними

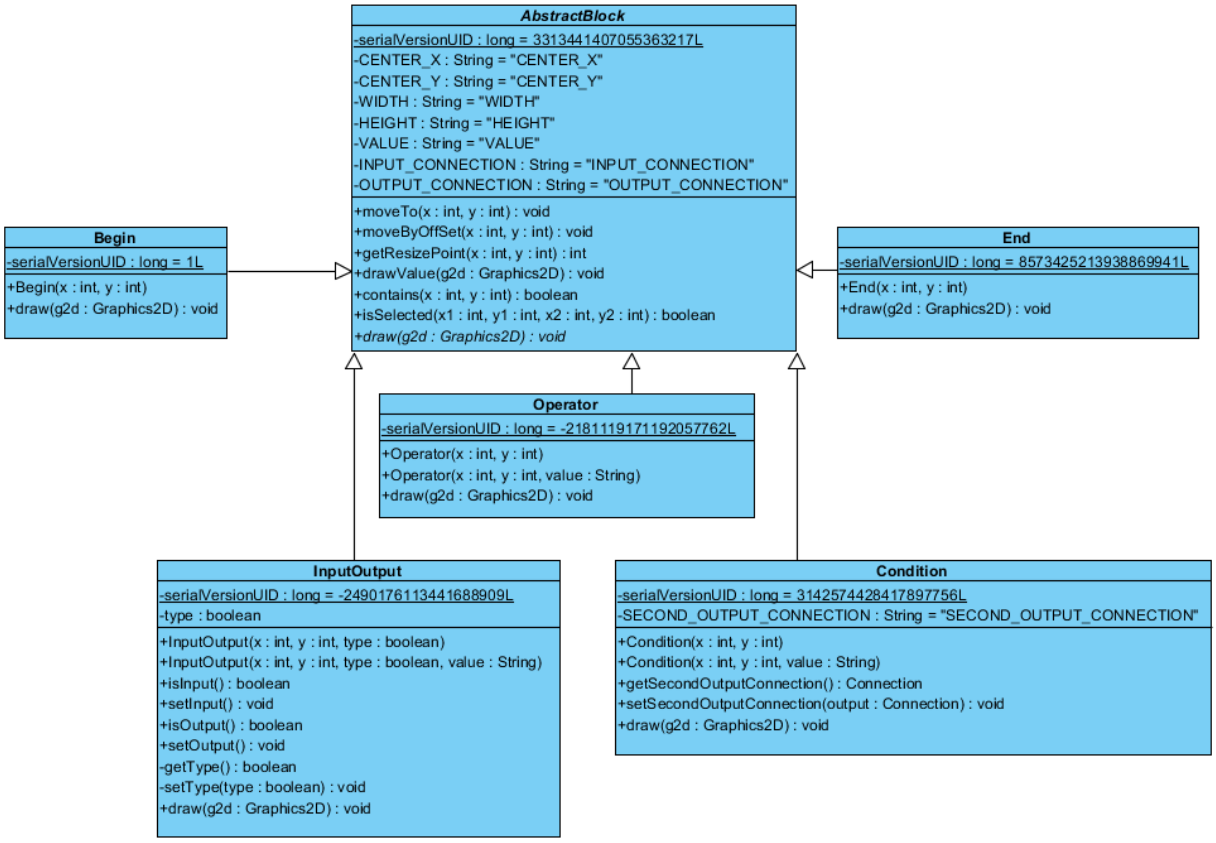

Рис. 2. Диаграмма классов, представляющих узлы ГСА

Подсистема проверки корректности построения блок-схемы

Блок-схема представляет собой дерево, начинающееся от терминальной вершины «Начало» и заканчивающееся терминальной вершиной «Конец».

При составлении графической схемы алгоритма могут иметь место следующие структурные ошибки:

- отсутствие терминальных вершин;
- множественность терминальных вершин одного класса;

- наличие бесконечных циклов (частично, в той мере, в какой оно определяется структурой блок-схемы);
- недостижимость вершины ГСА.

Проверка корректности структуры ГСА выполняется алгоритмом, описываемым с помощью диаграммы активности, приведенной на рис. 3.

Проверка наличия терминальных вершин, а также поиск «висячих вершин», не соединенных ни с одной из терминальных, осуществляется простой проверкой значений соответствующих переменных (задающих начальную и конечную вершину ГСА в первом случае и предыдущей и следующей вершины на последнем этапе). Диаграмма активности поиска вершин, недостижимых из текущей, приведена на рис. 4.

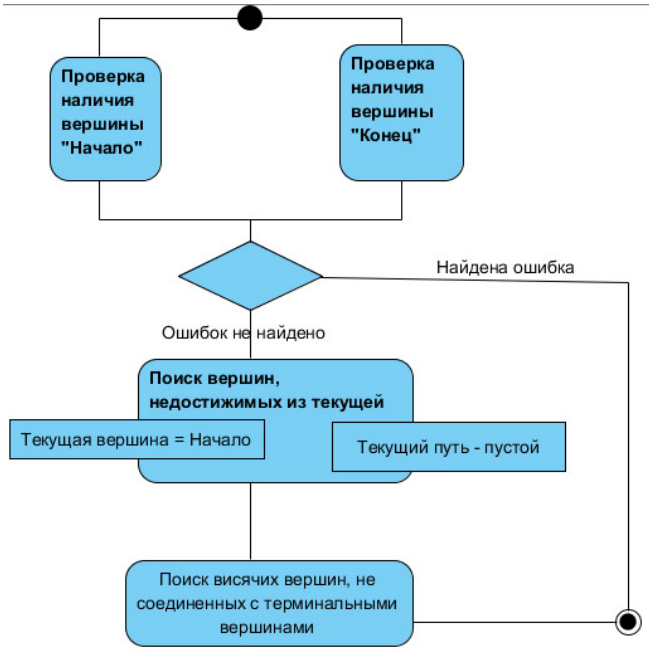

Рис. 3. Диаграмма активности при проверке корректности структуры ГСА

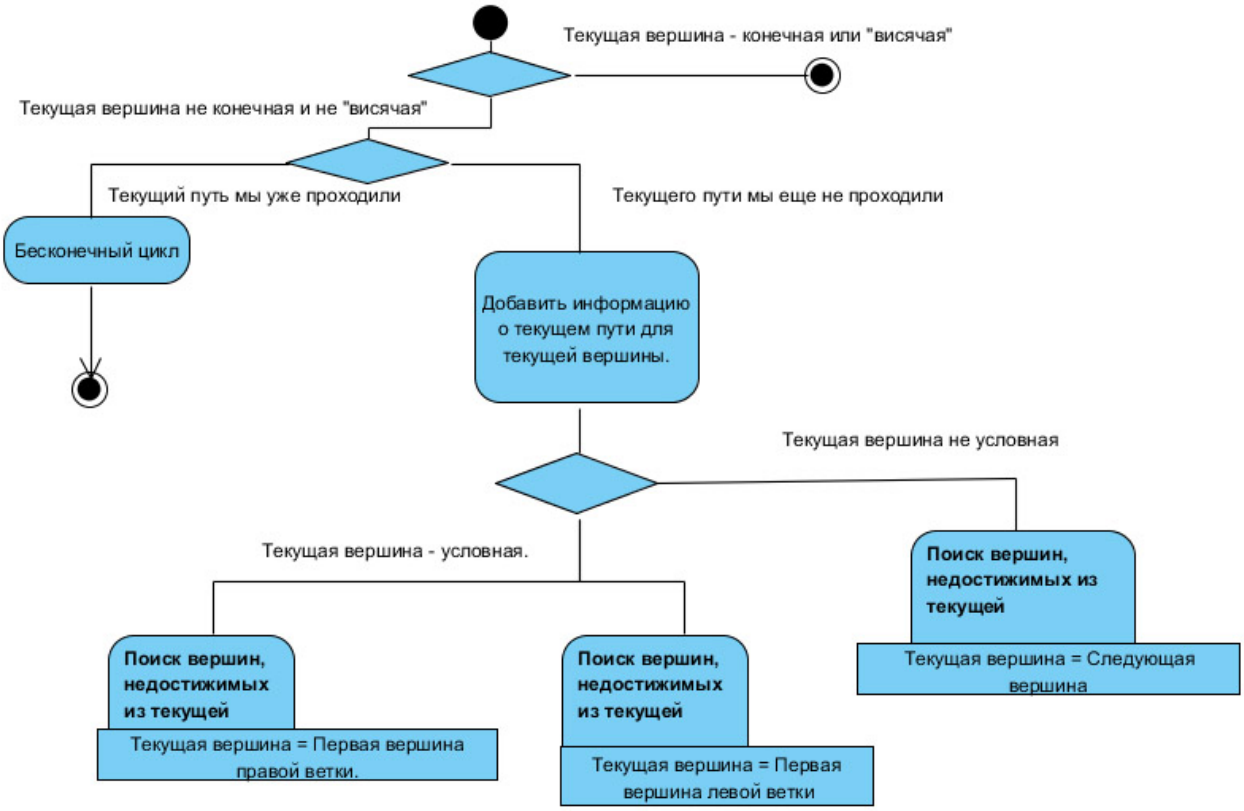

Рис. 4. Диаграмма активности поиска вершин, недостижимых из текущей

В соответствии с диаграммами активности проверка корректности ГСА сводится к алгоритму с возвратом поиска путей из каждой вершины в конечную. При отсутствии такого пути определяется количество бесконечных циклов и «висячие вершины».

Подсистема синтаксического и лексического анализа. Принципы программной реализации автоматов синтаксического и лексического разбора

Лексический и синтаксический анализ оператора реализован с помощью нескольких детерминированных конечных автома-

тов разного уровня. Автоматы задаются с помощью четырех таблиц, специфицированных в программе с помощью интерфейса Initiator:

- таблица классификации входных символов;
- таблица переходов ДКА;
- таблица выполняемых действий;
- таблица классификации возможных ошибок.

В таблице классификации входных символов каждому входному ASCII символу сопоставляется номер класса. Символы входного алфавита разделяются на классы, при этом символы одного класса одинаково обрабатываются.

Таблица переходов ДКА представляет собой массив целых чисел, задающий функцию переходов автомата. Двум аргументам (текущему состоянию и классу входного символа) сопоставляется следующее состояние и выполняемое действие. При этом если номер следующего состояния равен длине таблицы, это означает корректное завершение работы автомата, а в случае, когда он превышает длину таблицы, разность между ними определяет номер ошибки в таблице исключительных ситуаций.

Таблица выполняемых действий представляет собой массив экземпляров объектов, которые выполняют преобразования над элементами дерева синтаксического разбора. Также действием может быть предусмотрен переход в подавтомат.

Таблица классификации возможных исключительных ситуаций содержит экземпляры класса Exception. В случае обнаружения ошибки во входной последовательности автомат при помощи механизма исключительных ситуаций «выбрасывает» соответствующую ошибку, которая затем может быть обработана на уровне семантического анализа для информирования пользователя и попытки ее исправить.

В системе генерации кодов предусмотрено четыре конечных автомата разбора: автомат анализа операторов, автомат анализа имен переменных и функций, ДКА разбора числовых значений и автомат разбора текстовых констант.

Программная последовательность для реализации работы ДКА, заданного при помощи вышеуказанных четырех таблиц, определена в классе Thinker. Метод parse этого класса возвращает в качестве результата вершину дерева синтаксического разбора оператора. Этот класс имеет ссылку на экземпляр модели алгоритма, что позволяет обеспечить уникальность вводимых в дерево синтаксического разбора переменных.

Также введен класс StatementParser, приспособленный для поэтапного анализа значений всех операторных и условных блоков в ГСА. Он содержит дополнительные методы, связанные с корректным добавлением новых узлов в дерево синтаксического разбора.

Целевая структура данных синтаксического и лексического анализа

Абстракцией узла дерева синтаксического и лексического разбора является класс Operand. Операндами являются целое число, действительное число, строка, переменная, функция и оператор. Операнд имеет строчное значение, которое сформировано в результате работы автомата анализа. В дальнейшем, при генерации кодов, разнообразие видов узлов облегчит трансляцию на соответствующий язык программирования.

Экземпляры класса Function в поле значения хранят имя функции и ссылку на аргумент, также являющийся узлом дерева разбора.

В поле значения экземпляров класса Statement содержится название оператора. Также этот объект имеет связи с левой и правой частями оператора, в свою очередь являющихся узлами дерева лексико-синтаксического разбора.

Связь между элементами дерева разбора реализована посредством непосредственных ссылок на операнды, хранимых в узлах дерева.

Автомат разбора оператора

Детерминированный конечный автомат разбора оператора предназначен для корректного добавления новых операций в дерево разбора по следующим правилам:

- операция присваивания устанавливается в качестве начальной вершины дерева разбора. В дереве может быть лишь один оператор присваивания;
- остальные операции разделяются на уровни. Операция i-го, более высокого уровня, чем текущая, занимает в дереве разбора

место в правой ветке операции (i–1)-го уровня, при этом оператор, ранее стоявший в этом месте, становится левой веткой операции i-го уровня. В случае добавлении операции более низкого уровня, чем текущая, эта операция добавляется в правую ветку текущей.

Разделение операций по уровням показано в таблице 1.

Табл. 1. Разделение математических и логических операций по уровням

Уровень операции	Операция
0	= (присваивание)
1	| (логическое ИЛИ)
2	& (логическое И)
3	< (меньше) > (больше)
4	+ (сложение) – (вычитание)
5	/ (деление) * (умножение) % (деление по модулю)

Пример дерева лексико-синтаксического анализа для оператора b = a+c+e > f+d | k&p представлен на рис. 5.

Автомат разбора операторов программно реализуется при помощи класса StatementParser, для инициализации которого используется класс ParseStatement. Граф автомата разбора операторов представлен на рис. 6.

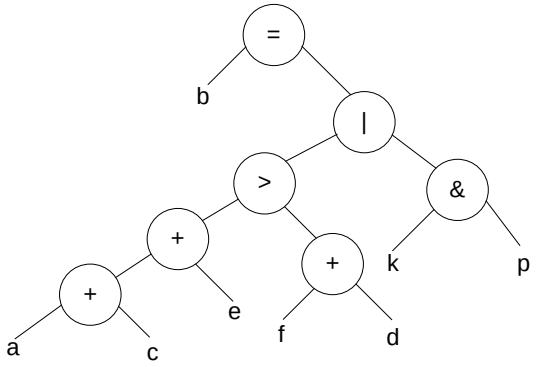

Рис. 5. Пример дерева лексико-синтаксического разбора

Автомат разбора имен переменных и функций

Данный автомат предназначен для добавления в дерево разбора переменных и функций. Имя переменной или функции начинается с буквы и может содержать буквы, цифры, символы подчеркивания «_», точку и квадратные скобки. Функция отличается от переменной тем, что содержит круглые скобки, содержащие оператор, который называется аргументом функции.

Когда автомат встречает другой символ во входной последовательности, он завершает свою работу и передает управление автомату более высокого уровня.

Граф ДКА анализа имен переменных и функций представлен на рис. 7.

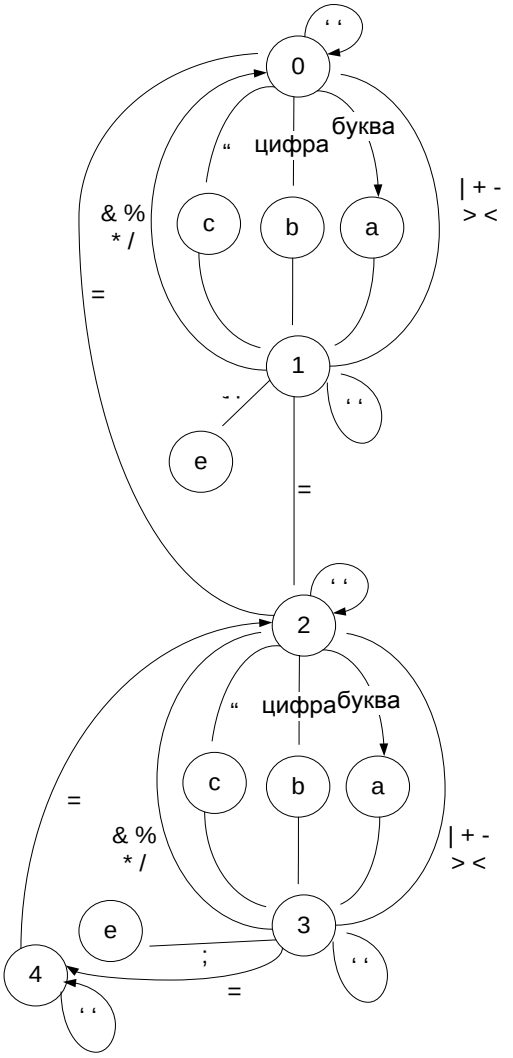

Рис. 6. Граф автомата разбора операторов a – переход к подавтомату разбора имен переменных и функций, b – переход с подавтомату разбора численных констант, c – переход к подавтомату разбора строчных констант, e – корректное завершение работы автомата.

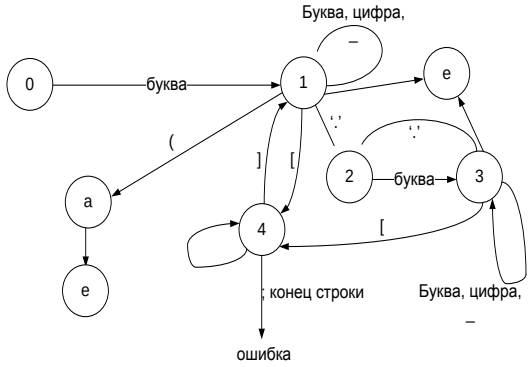

Рис. 7. Граф автомата разбора переменных и функций. a – переход к анализу аргументов функции, e – корректное завершение работы автомата.

Автомат разбора числовых значений

Числа – целые или действительные - начинаются с цифры. Разделителем между целой и дробной частью в действительном числе служит точка. Действительное число может начинаться с точки, в таком случае его целая часть равна нулю.

В случае, если автомат встречает другой символ (не цифру и не точку) он завершает свою работу и передает управление автомату более высокого уровня.

Числовые значения хранятся в соответствующих узлах дерева лексико-синтаксического разбора в виде текста, однако работа данного подавтомата гарантирует возможность корректного преобразования его в целый либо вещественный числовой тип при помощи стандартных средств языка Java.

Граф автомата-анализатора числовых констант представлен на рис. 8.

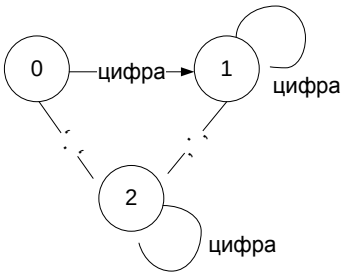

Рис.8. Граф автомата разбора числовых констант.

Автомат разбора строчных констант

Строчная константа начинается и заканчивается двойными кавычками. Внутри строчной константы разрешено использовать любые символы, за исключением комбинаций с символом «обратный слеш» («\»), не являющихся служебными символами. Служебными являются комбинации «\n» (переход на новую строку), «\t» (табуляция), «\"» (двойные кавычки) «\\»(обратный слеш).

Граф переходов детерминированного конечного автомата анализа строчных констант представлен на рис. 9.

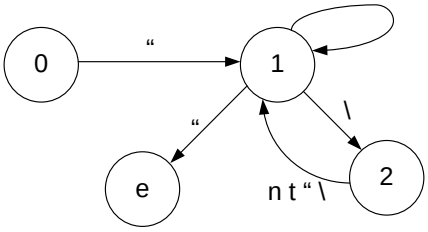

Рис. 9. Граф автомата разбора строчных констант.

Подсистема генерации кодов

Этапу генерации кода предшествуют этапы построения блок-схемы (расположение блоков, их связывание и указание значений) и верификации (проверка корректности структуры блок-схемы, лексический и синтаксический анализ значений операторов). В случае корректного прохождения этапа проверки в каждом операторном и условном блоке формируется дерево лексико-синтаксического разбора, а модель алгоритма содержит список всех используемых переменных. С этого момента начинается этап генерации кода. Вначале пользователю предлагается ввести типы используемых данных в специальном окне. После этого происходит формирование части программного кода с указанием используемых переменных и их типов. Затем осуществляется формирование тела программы путем обхода дерева блок-схемы. На этом этапе важно выделить в блок-схеме три различные структуры: цикл с предусловием (рис.10), цикл с постусловием (рис.11) и обычное ветвление (рис.12).

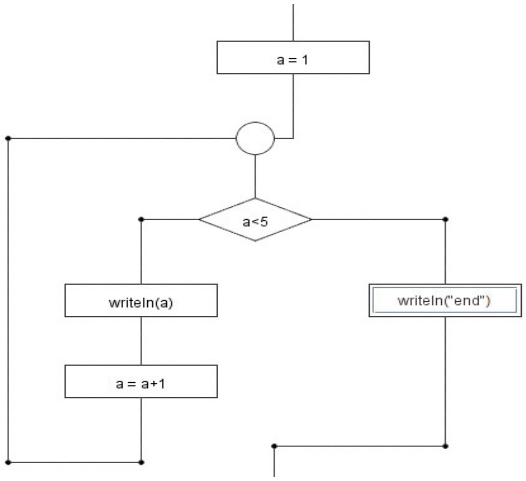

Рис. 10. Пример цикла с предусловием

Аналізуючи результати дослідження можна прослідкувати деяку схожість з теоретичними даними. Завдяки проведеним розра-

хункам і побудові графіка, що зображений вище, ми довели, що час роботи програми залежить не лише від кількості вузлів у топології, але й від кількості ребер, що з'єднують ці вузли.

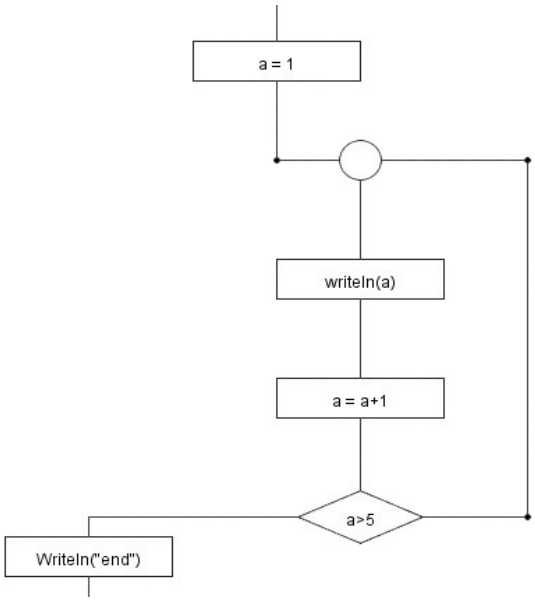

Рис.11. Пример цикла с постусловием

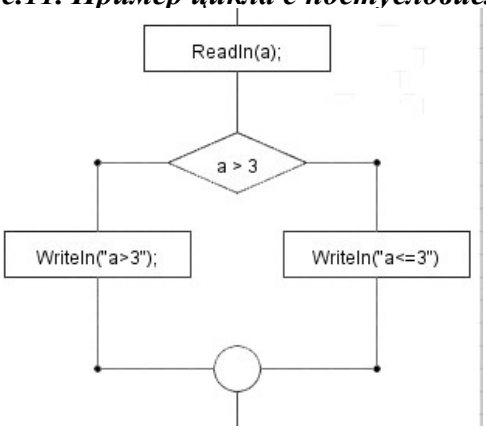

Рис 12. Пример нециклического ветвления

В случае, если текущая вершина – операторная, в код программы добавляется ее значение, оттранслированное на заданный язык программирования.

В том случае, если текущая вершина – предикатная (рис. 12), используется метод, возвращающий узел слияния, в котором сходятся обе ветки данной условной вершины. В текст программы добавляется оператор нециклического ветвления. Затем дважды рекурсивно запускается метод «step» для левой и правой ветки условия, при этом в качестве конечной вершины обхода задается найденный узел слияния. Таким образом обеспечивается рассмотрение условия как целостной структуры. В дальнейшем обход продолжается с выходного соединения найденного узла слияния.

Если текущий блок является узлом слияния, это является признаком цикла. В случае, когда концом исходящего соединения узла слияния является предикат (см. рис. 10), данный цикл является циклом с предусловием. Если же предикат является началом еще не пройденного входного соединения узла слияния (рис. 11), определяется цикл с постусловием. В обоих случаях для обработки тела цикла рекурсивно запускается метод «step», и дальнейшая обработка начинается со следующей за циклом вершины, что обеспечивает формирование цикла как единой структуры.

Разработанная система может быть использована как инструмент обучения программистов начального уровня, а также, при некоторой модификации, как дополнение к существующим CASE-системам для генерации внутреннего последовательного кода методов.

Список литературы

1. Р. Лингер, Х. Миллс, Б. Уитт «Теория и практика структурного программирования», М.: «Мир», 1982 – с. 99.
2. http://office.microsoft.com/
3. http://www.koffice.org/kivio/
4. http://tinyelectronic.org.ua/sites/default/files/programs/block-schem.exe
5. http://vicking.narod.ru/flowchart/index.html
6. http://alglib.manual.ru/blseditor.zip
7. Р. У. Себеста «Основные концепции языков программирования» - М.: "Вильямс", 2001 – с. 179.

Поступила в редакцию 17.12.2009